AI TOOLS FOR ACTUARIES
Chapter 8: Convolutional Neural Network - Part B

Ronald Richman, Salvatore Scognamiglio, Mario V. Wüthrich

2026-04-16

1. Convolutional neural networks
2. 1D convolutional neural networks
3. 2D convolutional neural networks
4. Deep convolutional neural networks
5. Appendix: General purpose layers

Convolutional neural networks

Overview

- This notebook introduces *convolutional neural networks* (CNNs).
- CNNs operate like *rolling windows*, moving a filter (window) across a time-series or an image to extract *local structure*.
- CNNs differ from FNNs in two key aspects:
  - *Local connectivity*: each neuron connects only to a small region (window) of the input, and not to the entire input.
  - *Parameter sharing*: the same set of weights (filter) is applied across all different regions of the input.
- Local connectivity and parameter sharing significantly reduces the dimension of the network parameter compared to FNN layers.

- CNNs were introduced by LeCun and Bengio (1998).

Recap: tensor input data

- Input data to networks is usually in tensor form.
- For single instances, the input information is either:
  - a vector $\boldsymbol{X} \in \mathbb{R}^q$ (1D tensor) for tabular input data,
  - a matrix $\boldsymbol{X}_{1:t} \in \mathbb{R}^{t \times q}$ (2D tensor) for time-series and text data,
  - a 3D tensor $\boldsymbol{X}_{1:t,1:s} \in \mathbb{R}^{t \times s \times q}$ for images and spatial data.
- The last dimension denotes the different channels having q components. The preceding dimensions have a time-series (2D) or a spatial (3D) structure that should be recognized by the network decoder.
- These input formats have been described in the previous notebook.

Types of network layers

When data is represented in a 2D or 3D tensor form, applying traditional FNN layers can be inappropriate:

- FNNs ignore time-series and/or spatial structure in the data,
- FNNs require a large number of parameters,
- FNNs cannot deal with time-series observations that are increasing over time.

To solve these issues, specialized network architectures have been designed. This notebook introduces

- *convolutional neural networks* (CNNs), and
- *locally-connected networks* (LCNs).

Subsequent notebooks discuss *recurrent neural networks* (RNNs) and *Transformers*.

Dimension of CNN layers

- CNN layers of different dimensions are available, e.g., there are 1D and 2D CNN layers.
- These CNN layers differ in the number of dimensions over which the convolution operation (local filter) is applied to:
  - for 2D tensors (time-series, text data) one uses a 1D CNN layer;
  - for 3D tensors (images, spatial data) one often uses a 2D CNN layer.
- Example: A RGB image has a 2D spatial structure and $q = 3$ color channels resulting in a 3D tensor

$$\boldsymbol{X}_{1:t,1:s} = (X_{u,v,j})_{1 \leq u \leq t,\, 1 \leq v \leq s,\, 1 \leq j \leq 3} \in \mathbb{R}^{t \times s \times 3}.$$

  To process this 3D tensor we typically use a 2D CNN layer to *respect* the 2D spatial structure.

1 Convolutional neural networks

2 1D convolutional neural networks

3 2D convolutional neural networks

4 Deep convolutional neural networks

5 Appendix: General purpose layers

1D convolutional neural networks

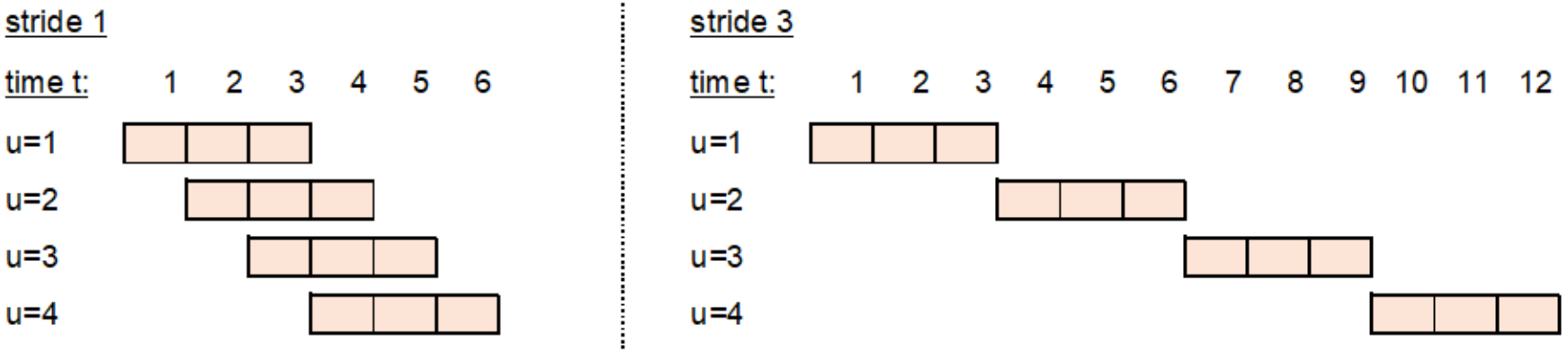
Overview

- 1D CNNs are specifically designed to handle data of *time-series structure*, such as sequences and text data.
- 1D CNN architectures apply a *convolution operation* along the time axis – the positional index.
- 1D CNN filters act as rolling windows along the time axis over the input data extracting *local structure*.
- This respects the time-causal structure.

Hyper-parameters of a 1D CNN layer

- A 1D CNN layer involves the following two hyper-parameters:
  - *Kernel size* $K \in \mathbb{N}$: defines the length (window size) of the *filter* used in the convolution operation.
  - *Stride* $\delta \in \mathbb{N}$: specifies the step size of the filter movement (translation).

Parameters K and δ balance *computational cost* and *feature resolution*:



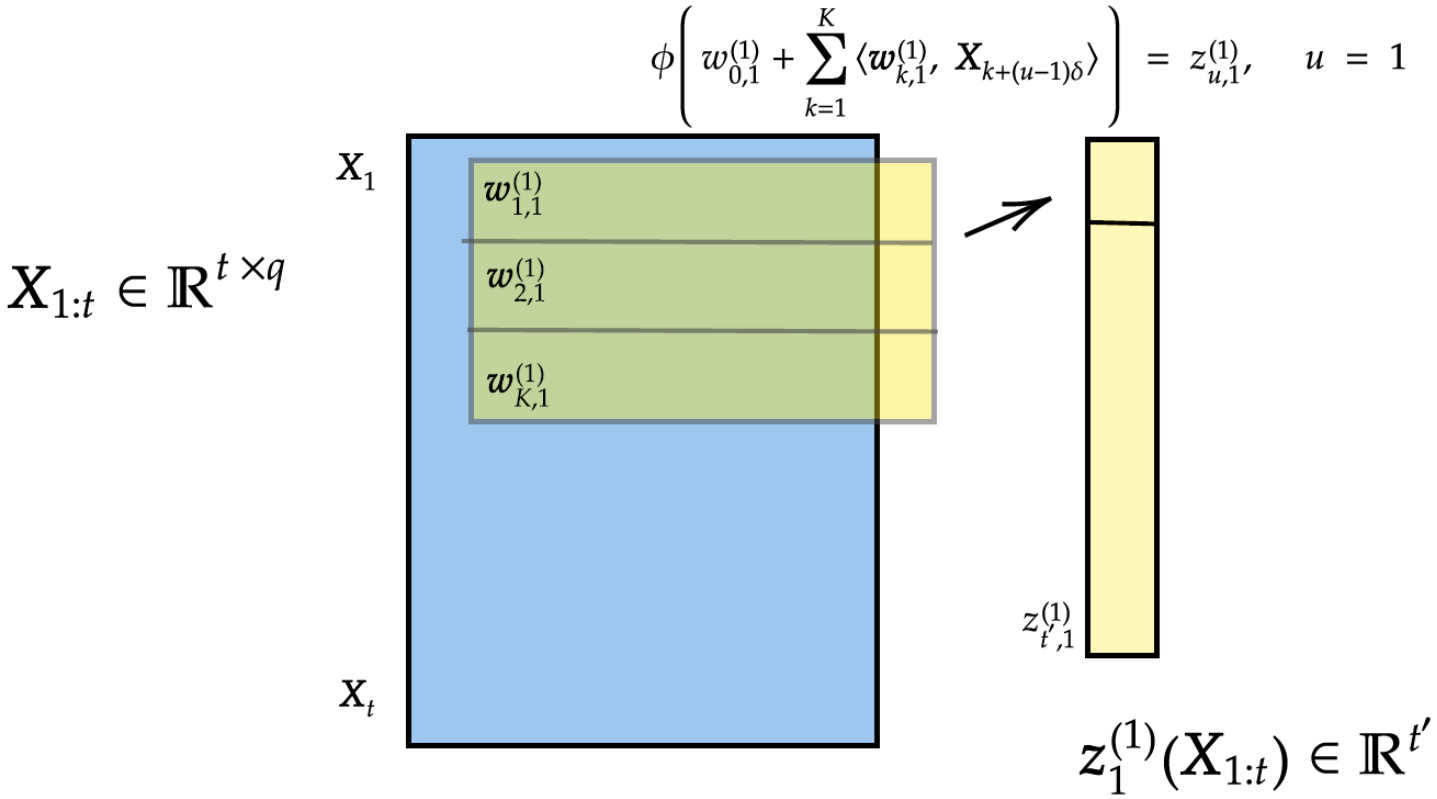
The filter of kernel size $K = 3$ moves along the (horizontal) time axis $t \geq 1$ with stride $\delta = 1$ (lhs) and stride $\delta = 3$ (rhs).

1D CNNs: single filter

- Let $\boldsymbol{X}_{1:t} \in \mathbb{R}^{t \times q}$ be the input matrix (2D tensor) with q channels.
- A 1D CNN layer with a *single filter* of size K and stride δ is a mapping

$$\begin{aligned} \boldsymbol{z}_1^{(1)} \quad : \quad & \mathbb{R}^{t \times q} \to \mathbb{R}^{t'} = \mathbb{R}^{t' \times 1}, \\ & \boldsymbol{X}_{1:t} \mapsto \boldsymbol{z}_1^{(1)}(\boldsymbol{X}_{1:t}) = \left(z_{u,1}^{(1)}(\boldsymbol{X}_{1:t}) \right)_{1 \le u \le t'}, \end{aligned}$$

  with $t' = \lfloor \frac{t-K}{\delta} + 1 \rfloor \in \mathbb{N}$ representing the number of *receptive fields*.
- The units $z_{u,1}^{(1)}(\boldsymbol{X}_{1:t}) \in \mathbb{R}$ are defined after the next plot, which illustrates their construction.
- The lower index $_1$ (filters) and the upper index $^{(1)}$ (depth/layers) will become clear below.



- The filter of dimension $K \times q$ projects all the values using the yellow window to a single value $z_{u,1}^{(1)}(\boldsymbol{X}_{1:t}) \in \mathbb{R}$, here $u = 1$.
- This projects the number of channels q down to 1 (horizontal axis).
- The length of the time-series shortens by the kernel size K and stride δ from t to $t' = \lfloor \frac{t-K}{\delta} + 1 \rfloor$ (vertical axis).

1D CNNs: convolutional units

- A 1D CNN layer with a *single filter* of size K and stride δ is a mapping

$$\begin{aligned} \boldsymbol{z}_1^{(1)} \quad : \quad & \mathbb{R}^{t \times q} \to \mathbb{R}^{t'} = \mathbb{R}^{t' \times 1}, \\ & \boldsymbol{X}_{1:t} \mapsto \boldsymbol{z}_1^{(1)}(\boldsymbol{X}_{1:t}) = \left(z_{u,1}^{(1)}(\boldsymbol{X}_{1:t}) \right)_{1 \leq u \leq t'}, \end{aligned}$$

  with $t' = \lfloor \frac{t-K}{\delta} + 1 \rfloor \in \mathbb{N}$ representing the number of *receptive fields*.

- Unit $z_{u,1}^{(1)}(\boldsymbol{X}_{1:t})$ *convolves* the filter with the u-th receptive field

$$z_{u,1}^{(1)}(\boldsymbol{X}_{1:t}) = \phi \left(w_{0,1}^{(1)} + \sum_{k=1}^{K} \left\langle \boldsymbol{w}_{k,1}^{(1)}, \boldsymbol{X}_{(u-1)\delta+k} \right\rangle \right),$$

  with *bias* $w_{0,1}^{(1)} \in \mathbb{R}$, *filter weights* $\boldsymbol{w}_{k,1}^{(1)} \in \mathbb{R}^q$ and *activation* $\phi : \mathbb{R} \to \mathbb{R}$.

- This involves $1 + Kq$ parameters.

- We bring this into vector form

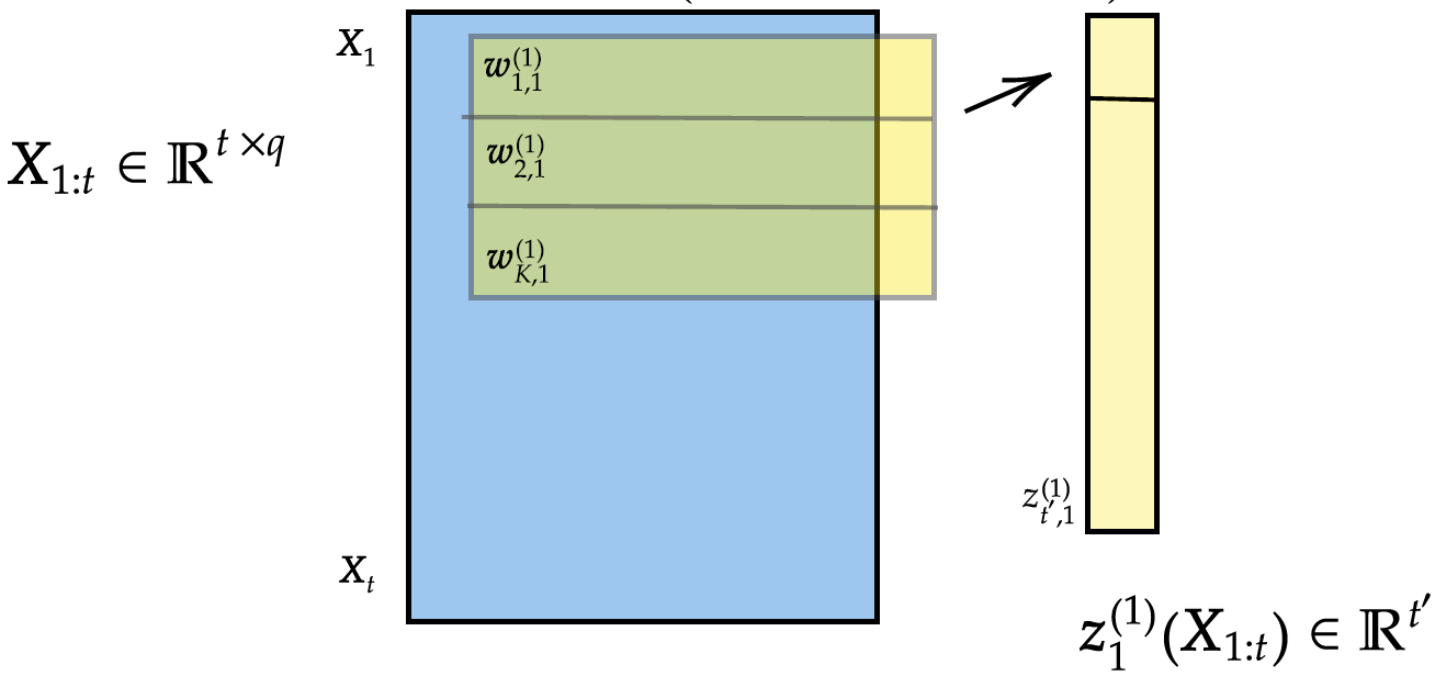
$$\boldsymbol{z}_1^{(1)}(\boldsymbol{X}_{1:t}) = \begin{pmatrix} z_{1,1}^{(1)}(\boldsymbol{X}_{1:t}) \\ \vdots \\ z_{t',1}^{(1)}(\boldsymbol{X}_{1:t}) \end{pmatrix} = \begin{pmatrix} \phi\left(w_{0,1}^{(1)} + \sum_{k=1}^{K} \left\langle \boldsymbol{w}_{k,1}^{(1)}, \boldsymbol{X}_{(1-1)\delta+k} \right\rangle\right) \\ \vdots \\ \phi\left(w_{0,1}^{(1)} + \sum_{k=1}^{K} \left\langle \boldsymbol{w}_{k,1}^{(1)}, \boldsymbol{X}_{(t'-1)\delta+k} \right\rangle\right) \end{pmatrix}$$

$$=: \phi\left(w_{0,1}^{(1)} + W_1^{(1)} * \boldsymbol{X}_{1:t}\right) \in \mathbb{R}^{t'},$$

with filter weights $W_1^{(1)} = [\boldsymbol{w}_{1,1}^{(1)}, \ldots, \boldsymbol{w}_{K,1}^{(1)}]^\top \in \mathbb{R}^{K \times q}$, and where the bias $w_{0,1}^{(1)}$ is added to every component $1 \leq u \leq t'$.

- The filter $W_1^{(1)} \in \mathbb{R}^{K \times q}$ moves along the time axis of $\boldsymbol{X}_{1:t}$ (with stride δ) – we abbreviate this by the convolution operator $*$ (having stride δ).
- The new time-series $\boldsymbol{z}_1^{(1)}(\boldsymbol{X}_{1:t}) \in \mathbb{R}^{t'}$ of length t' summarizes in every component u the u-th receptive field of the input time-series $\boldsymbol{X}_{1:t}$ – this precisely means *extracting local structure*.

$$\phi\left(w_{0,1}^{(1)} + \sum_{k=1}^{K} \langle \boldsymbol{w}_{k,1}^{(1)}, \mathbf{X}_{k+(u-1)\delta} \rangle\right) = z_{u,1}^{(1)}, \quad u = 1$$



- The filter with $K = 3$ applies the scalar products across all channels q.
- The filter moves down the time axis by $\delta = 1$ steps.
- The filter weights are shared across all receptive fields $1 \leq u \leq t'$.

1D CNNs: multiple filter

- Generally, a single filter is not sufficient to capture the complexity of the input data – this is equivalent to the number of neurons in a FNN layer.
- A 1D CNN layer with $q_1 \in \mathbb{N}$ filters is a mapping

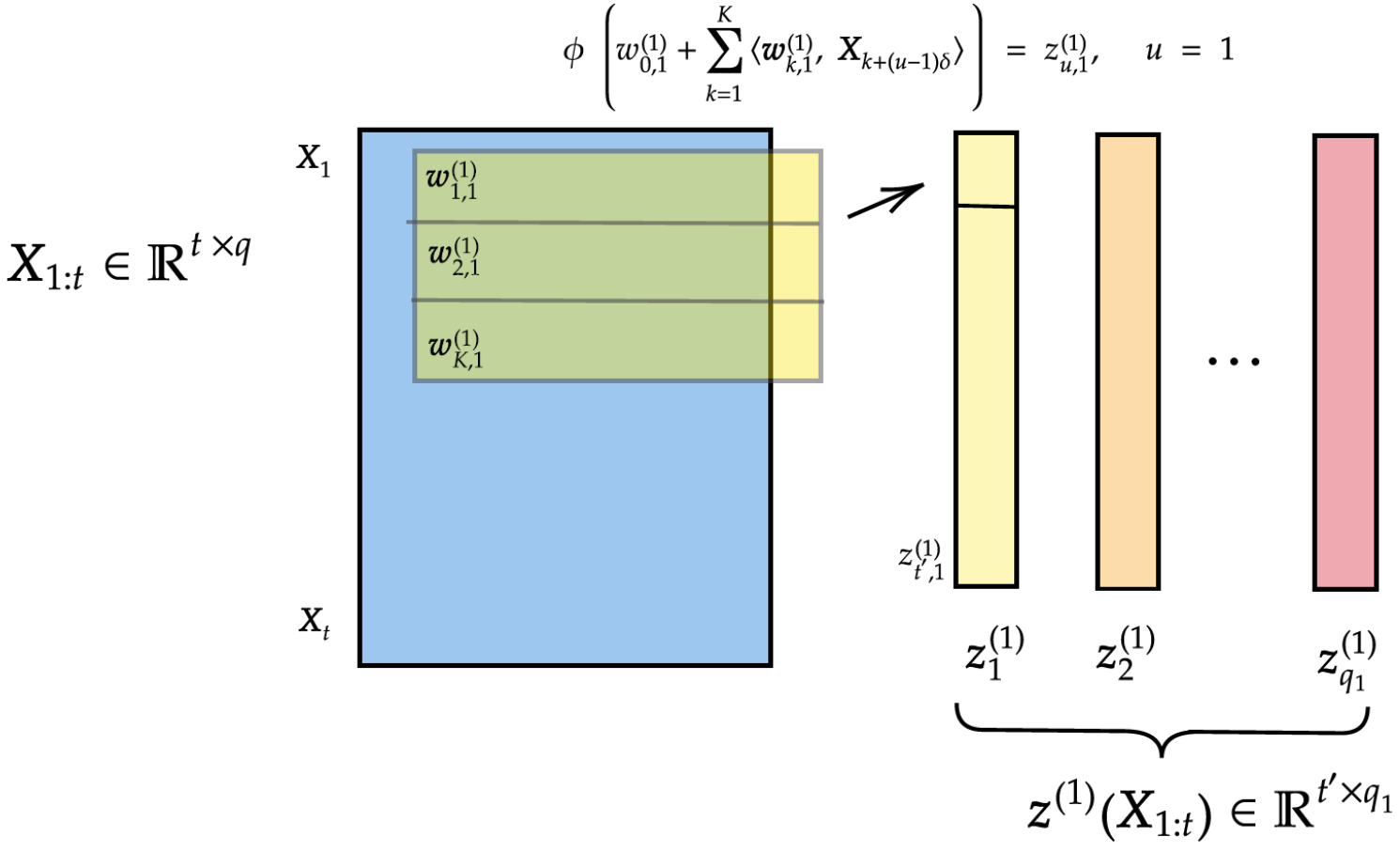
$$\begin{aligned} \boldsymbol{z}^{(1)} \quad : \quad & \mathbb{R}^{t \times q} \to \mathbb{R}^{t' \times q_1} \\ & \boldsymbol{X}_{1:t} \mapsto \boldsymbol{z}^{(1)}(\boldsymbol{X}_{1:t}) = \left(\boldsymbol{z}_1^{(1)}(\boldsymbol{X}_{1:t}), \ldots, \boldsymbol{z}_{q_1}^{(1)}(\boldsymbol{X}_{1:t}) \right), \end{aligned}$$

  with for $1 \leq j \leq q_1$

$$\boldsymbol{z}_j^{(1)}(\boldsymbol{X}_{1:t}) = \phi\left(w_{0,j}^{(1)} + W_j^{(1)} * \boldsymbol{X}_{1:t} \right) \in \mathbb{R}^{t'},$$

  for biases $w_{0,j}^{(1)} \in \mathbb{R}$, filter weights $W_j^{(1)} \in \mathbb{R}^{K \times q}$, and where the convolution operator $*$ has stride δ.
- This uses the lower index $_j$ in the notation.

Illustration of 1D CNN layer with multiple filters



- The output $\boldsymbol{z}^{(1)}(\boldsymbol{X}_{1:t}) \in \mathbb{R}^{t' \times q_1}$ is again a 2D tensor and time-causality along the vertical axis is respected by the rolling window mechanism.

- The j-th column of $\boldsymbol{z}^{(1)} \in \mathbb{R}^{t' \times q_1}$, containing the elements

$$\left(z_{1,j}^{(1)}, z_{2,j}^{(1)}, \ldots, z_{t',j}^{(1)}\right)^\top \in \mathbb{R}^{t'},$$

  represents a set of features extracted by applying the *same* filter to the different receptive fields - this is the time-causal part.

- The u-th row of $\boldsymbol{z}^{(1)} \in \mathbb{R}^{t' \times q_1}$, containing the elements

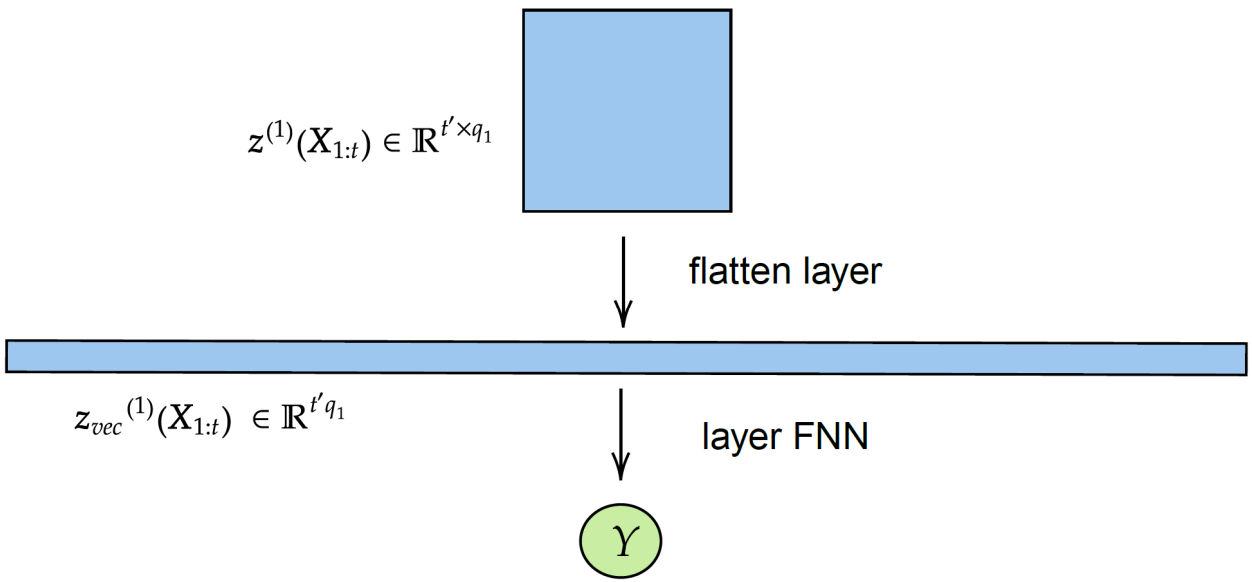
$$z_{u,1}^{(1)}, z_{u,2}^{(1)}, \ldots, z_{u,q_1}^{(1)},$$

  represents a set of features obtained by applying q_1 *different* filters to the same u-th receptive field - similar to FNN neurons, these are the different data compressions (representations) of the u-th receptive field.

- The total number of parameters to be learned in a 1D CNN layer with q_1 filters of size K is $(1 + Kq)q_1$.

Flatten layer

- A CNN outputs a tensor of the same order as the input. For time-series input data, this is 2D tensor $\boldsymbol{z}^{(1)}(\boldsymbol{X}_{1:t}) \in \mathbb{R}^{t' \times q_1}$.
- To use $\boldsymbol{z}^{(1)}(\boldsymbol{X}_{1:t})$ for prediction (e.g., forecasting an insurance claim), further processing is needed by decoding it, e.g., with a FNN layer.
- This requires flattening the 2D tensor $\boldsymbol{z}^{(1)}(\boldsymbol{X}_{1:t}) \in \mathbb{R}^{t' \times q_1}$ into a vector $\boldsymbol{z}^{(1)}_{\text{vec}}(\boldsymbol{X}_{1:t}) \in \mathbb{R}^{t' q_1}$ of length $t' q_1$ using a so-called *flatten layer*.



Example of a 1D CNN

The following code illustrates a network architecture containing a single 1D CNN layer.

The `CNN1D_Model` function takes the following arguments:

- `seed`: to ensure reproducibility of results;
- `input_size`: input dimensions (sequence length t and number of channels q);
- `filters`: number of CNN filters q_1;
- `kernel_size`: size of the CNN kernel K;
- `stride`: stride for the resolution δ;
- `output_size`: output dimension (of forecast).

The model below includes a 1D CNN layer, followed by a flatten layer and a decoding FNN layer to compute the output.

```
# CNN_1D model function
library(keras); library(tensorflow)

CNN1D_Model <- function(seed, input_size, filters, kernel_size, stride,
↪  output_size) {
      k_clear_session(); set.seed(seed); set_random_seed(seed)
      Design <- layer_input(shape = input_size, dtype = 'float32')

      # 1D CNN encoder and a FNN decoder
      Response <- Design %>%
           layer_conv_1d(filters = filters, stride = stride, kernel_size =
           ↪  kernel_size, activation = 'linear')  %>%
           layer_flatten() %>%
           layer_dense(output_size, activation = "linear")

      # model definition
      keras_model(inputs = c(Design), outputs = Response)
}
```

- This example shows how to create a 1D CNN model applied to a 2D input tensor of size $(t, q) = (10, 5)$. The model uses a single $q_1 = 1$ filter, a kernel size of $K = 3$, a stride $\delta = 1$, and 1 output unit.

```
# Example usage
model <- CNN1D_Model(
  seed = 100,
  input_size = c(10, 5),
  filters = 1,
  kernel_size = 3,
  stride = 1,
  output_size = 1)
```

- This architecture has 25 weights to be fitted, see next table below, and it uses linear activation functions (which could be replaced in the above code).

```
summary(model)
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 10, 5)]           0
 conv1d (Conv1D)             (None, 8, 1)              16
 flatten (Flatten)           (None, 8)                 0
 dense (Dense)               (None, 1)                 9
=================================================================
Total params: 25 (100.00 Byte)
Trainable params: 25 (100.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

- The number of CNN weights are $(1 + Kq) = 16$.
- A stride of $\delta = 1$ and a kernel size of $K = 3$ reduces the length $t = 10$ of the original time-series to $t' = 8$.

- This example shows how to create a 1D CNN model with multiple filters $q_1 = 4$, a kernel size of $K = 3$, a stride $\delta = 1$, and 1 output unit.

```
# Example usage
model <- CNN1D_Model(
  seed = 100,
  input_size = c(10, 5),
  filters = 4,
  kernel_size = 3,
  stride = 1,
  output_size = 1)
```

- This architecture has 97 weights to be fitted, see next table below, and it uses linear activation functions.
- We notice that the use of weights is rather economical because of parameter (filter) sharing along the time t axis.

```
summary(model)
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 10, 5)]           0
 conv1d (Conv1D)             (None, 8, 4)              64
 flatten (Flatten)           (None, 32)                0
 dense (Dense)               (None, 1)                 33
=================================================================
Total params: 97 (388.00 Byte)
Trainable params: 97 (388.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

- The number of CNN weights are $(1 + Kq)q_1 = 16 \cdot 4 = 64$.
- Again, a stride of $\delta = 1$ and a kernel size of $K = 3$ reduces the length $t = 10$ of the original time-series to $t' = 8$.

1 Convolutional neural networks

2 1D convolutional neural networks

3 2D convolutional neural networks

4 Deep convolutional neural networks

5 Appendix: General purpose layers

2D convolutional neural networks

Overview

- 2D CNNs are specifically designed to process data structured in a two-dimensional grid, such as images or spatially structured data sets.
- The input data consists of three dimensions and is represented as a 3D tensor in $\mathbb{R}^{t \times s \times q}$, where q represents the number of *channels*.
- The convolution operation involves sliding a window across the two axes t and s (the spatial dimensions), while simultaneously processing all q channels at each position.
- This enables the model to detect *local patterns* in the input data (through the window/kernel), and it respects the spatial topology.

Hyper-parameters of a 2D CNN layer

- For a 2D CNN layer, the modeler needs to specify hyper-parameters along both spatial directions.
- In a 2D CNN, both, kernel size and stride, are defined as pairs:
  - Kernel size $K = (K_t, K_s) \in \mathbb{N}^2$: K_t represents the *height of the kernel* (number of rows), and K_s the *width of the kernel* (number of columns).
  - Stride $\delta = (\delta_t, \delta_s) \in \mathbb{N}^2$: δ_t denotes the vertical stride (the number of rows the filter moves), and δ_s the horizontal stride (the number of columns the filter moves).
- Based on these choices 1D CNN layers and 2D CNN layers work rather similarly.

2D CNNs: single filter

- Let $\boldsymbol{X}_{1:t,1:s} \in \mathbb{R}^{t \times s \times q}$ be the 3D input tensor.
- A 2D CNN layer with a *single filter* of size (K_t, K_s) and stride (δ_t, δ_s) is a mapping

$$\begin{aligned} \boldsymbol{z}_1^{(1)} \quad : \quad & \mathbb{R}^{t \times s \times q} \to \mathbb{R}^{t' \times s'} = \mathbb{R}^{t' \times s' \times 1} \\ & \boldsymbol{X}_{1:t,1:s} \mapsto \boldsymbol{z}_1^{(1)}(\boldsymbol{X}_{1:t,1:s}) = \left(z_{u,v,1}^{(1)}(\boldsymbol{X}_{1:t,1:s}) \right)_{1 \le u \le t', 1 \le v \le s'} . \end{aligned}$$

- The output of a 2D CNN layer with a single filter is a matrix with $t' = \left\lfloor \frac{t - K_t}{\delta_t} + 1 \right\rfloor$ rows and $s' = \left\lfloor \frac{s - K_s}{\delta_s} + 1 \right\rfloor$ columns.

- Unit $z^{(1)}_{u,v,1}(\boldsymbol{X}_{1:t,1:s})$ is extracted by convolving the filter with the receptive field (u, v)

$$z^{(1)}_{u,v,1}(\boldsymbol{X}_{1:t,1:s}) = \phi\left(w^{(1)}_{0,1} + \sum_{k_t=1}^{K_t}\sum_{k_s=1}^{K_s}\left\langle \boldsymbol{w}^{(1)}_{k_t,k_s,1}, \boldsymbol{X}_{(u-1)\delta_t+k_t,(v-1)\delta_s+k_s}\right\rangle\right),$$

with bias $w^{(1)}_{0,1} \in \mathbb{R}$ and filter weights $\boldsymbol{w}^{(1)}_{k_t,k_s,1} \in \mathbb{R}^q$, for $1 \le k_t \le K_t$ and $1 \le k_s \le K_s$.

- We abbreviate this by the convolution form (similarly to above)

$$\boldsymbol{z}^{(1)}_1(\boldsymbol{X}_{1:t,1:s}) = \phi\left(w^{(1)}_{0,1} + W^{(1)}_1 * \boldsymbol{X}_{1:t,1:s}\right) \in \mathbb{R}^{t'\times s'},$$

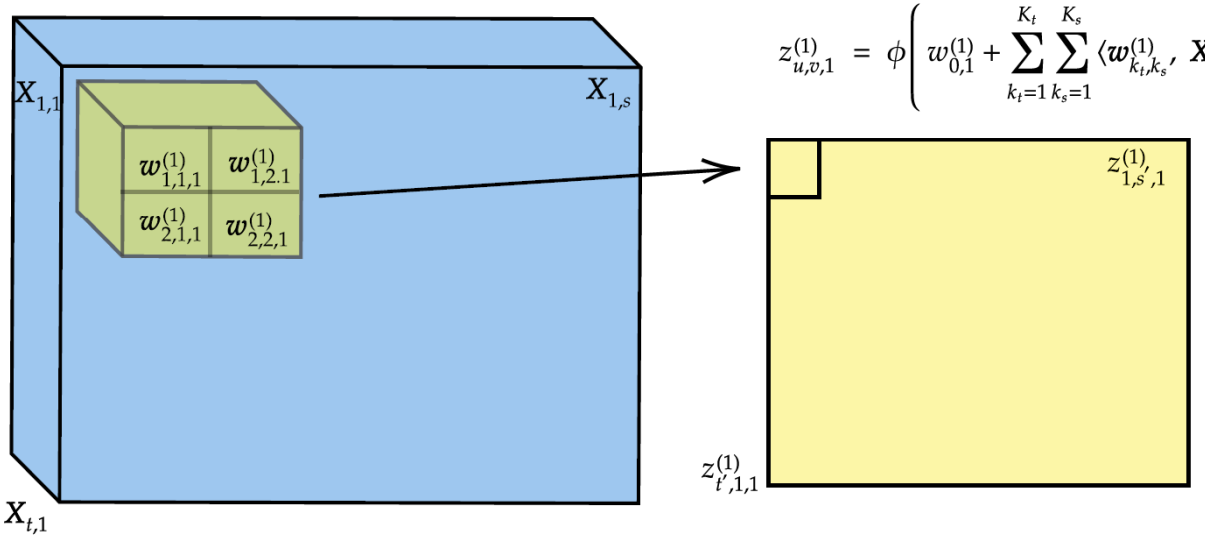
with filter weights

$$W^{(1)}_1 = \left(\boldsymbol{w}^{(1)}_{k_t,k_s,1}\right)_{1\le k_t\le K_t, 1\le k_s\le K_s} \in \mathbb{R}^{K_t\times K_s\times q},$$

the convolution operation $*$ has stride (δ_t, δ_s), and the bias $w^{(1)}_{0,1} \in \mathbb{R}$ is added to every component.

Illustration of 2D CNN filter

$$\mathbf{X}_{1:t,1:s} \in \mathbb{R}^{t \times s \times q}$$



$$z^{(1)}_{u,v,1} = \phi\left(w^{(1)}_{0,1} + \sum_{k_t=1}^{K_t} \sum_{k_s=1}^{K_s} \langle \boldsymbol{w}^{(1)}_{k_t,k_s}, \mathbf{X}_{k_t+(u-1)\delta_t,k_s+(v-1)\delta_s} \rangle \right), \; u, v = 1$$

$$\boldsymbol{z}^{(1)}_1(\mathbf{X}_{1:t,1:s}) \in \mathbb{R}^{t' \times s'}$$

- The rolling window (filter) of size $K_t \times K_s \times q$ now moves across both spatial axes.
- This respects the spatial topology.

2D CNN layer: multiple filters

- A 2D CNN layer with multiple filters is obtained by the intuitive extension

$$\begin{aligned} \boldsymbol{z}^{(1)} \quad : \quad & \mathbb{R}^{t \times s \times q} \to \mathbb{R}^{t' \times s' \times q_1} \\ & \boldsymbol{X}_{1:t,1:s} \mapsto \boldsymbol{z}^{(1)}(\boldsymbol{X}_{1:t,1:s}) = \left(\boldsymbol{z}_j^{(1)}(\boldsymbol{X}_{1:t,1:s}) \right)_{1 \leq j \leq q_1}, \end{aligned}$$

  where the different filters use biases $w_{0,j}^{(1)} \in \mathbb{R}$ and weights $W_j^{(1)} \in \mathbb{R}^{K_t \times K_s \times q}$, $1 \leq j \leq q_1$.

- We refrain from giving more technical details, but things are completely analogous to the 1D CNN case.

1 Convolutional neural networks

2 1D convolutional neural networks

3 2D convolutional neural networks

4 Deep convolutional neural networks

5 Appendix: General purpose layers

Deep convolutional neural networks

Overview

- Multiple CNN layers can be composed to capture increasingly complex and hierarchical features from the input data.
- As standard in network architectures, the output of one layer serves as input to the next layer, allowing information to propagate through the network.
- Each CNN layer is characterized by a specific set of hyper-parameters such as the kernel size and the stride, which can vary across layers to optimally extract feature information.
- We now use the upper index in the notation $\boldsymbol{z}^{(m)}$.

- W.l.o.g., we present the case of a deep 2D CNN architecture.

Deep CNN architectures

- Select a first 2D CNN layer

$$\boldsymbol{z}^{(1)} : \mathbb{R}^{t \times s \times q} \to \mathbb{R}^{t' \times s' \times q_1},$$

with kernel size $K^{(1)}$ and stride $\delta^{(1)}$.

- Select a second 2D CNN layer

$$\boldsymbol{z}^{(2)} : \mathbb{R}^{t' \times s' \times q_1} \to \mathbb{R}^{t'' \times s'' \times q_2},$$

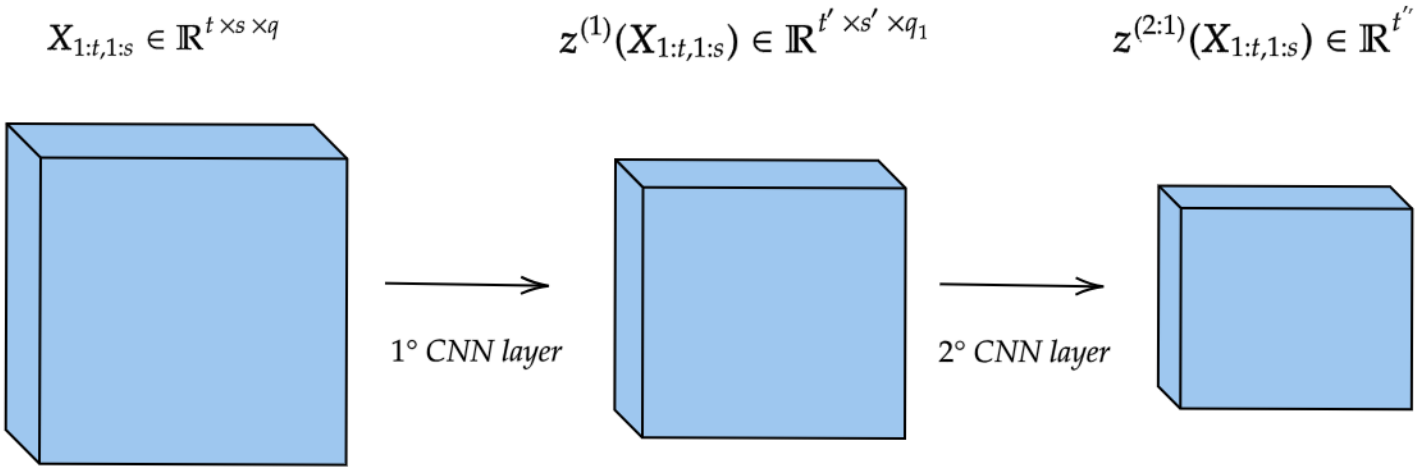
with new spatial dimensions

$$t'' = \left\lfloor \frac{t' - K_t^{(2)}}{\delta_t^{(2)}} + 1 \right\rfloor \in \mathbb{N} \quad \text{and} \quad s'' = \left\lfloor \frac{s' - K_s^{(2)}}{\delta_s^{(2)}} + 1 \right\rfloor \in \mathbb{N},$$

for kernel size $K^{(2)}$ and stride $\delta^{(2)}$.

- Compose these two 2D CNN layers to a *deep 2D CNN architecture* by composing

$$\boldsymbol{z}^{(2:1)} : \mathbb{R}^{t \times s \times q} \to \mathbb{R}^{t'' \times s'' \times q_2} \qquad \text{by} \qquad \boldsymbol{z}^{(2:1)} = \boldsymbol{z}^{(2)} \circ \boldsymbol{z}^{(1)}.$$


$X_{1:t,1:s} \in \mathbb{R}^{t \times s \times q}$
$z^{(1)}(X_{1:t,1:s}) \in \mathbb{R}^{t' \times s' \times q_1}$
$z^{(2:1)}(X_{1:t,1:s}) \in \mathbb{R}^{t'' \times s'' \times q_2}$
1° CNN layer
2° CNN layer

- This can be generalized to any depth $d \geq 2$.

Example: deep CNN architecture

The following code gives a function `Deep_CNN2D_Model` which builds a 2D CNN architecture of depth 2.

The most important arguments of this function are:

- `filters`: a list specifying the number of filters for each of the two CNN layers;
- `kernel_size`: a list specifying the kernel sizes for each CNN layer;
- `stride`: a list indicating the stride values for each CNN layer.

```
Deep_CNN2D_Model <- function(seed, input_size, filters, kernel_size, stride,
↪  output_size) {
      k_clear_session(); set.seed(seed); set_random_seed(seed)
      # inputs: input_size should be c(height, width, channels)
      Design <- layer_input(shape = input_size, dtype = 'float32')

      # deep 2D CNN encoder
      Response <- Design %>%
        layer_conv_2d(filters = filters[[1]], kernel_size = kernel_size[[1]],
        ↪  strides = stride[[1]], activation = 'relu') %>%
        layer_conv_2d(filters = filters[[2]], kernel_size = kernel_size[[2]],
        ↪  strides = stride[[2]], activation = 'relu') %>%
        # FNN decoder
        layer_flatten() %>%
        layer_dense(units = output_size, activation = 'linear')

      # model definition
      keras_model(inputs = Design, outputs = Response)
}
```

- The following code creates a deep architecture containing two 2D CNN layers with filters $q_1 = 4$ and $q_2 = 2$, kernel sizes $K^{(1)} = (3, 3)$ and $K^{(2)} = (4, 4)$ and stride values $\delta^{(1)} = \delta^{(2)} = (1, 1)$:

```
# Example usage
model <- Deep_CNN2D_Model(
    seed = 100,
    input_size = c(10, 10, 5),
    filters = list(4, 2),
    kernel_size = list(c(3, 3), c(4, 4)),
    stride = list(c(1, 1), c(1, 1)),
    output_size = 1
)
```

- This deep 2D CNN architecture has 365 weights to be fitted.

```
summary(model)
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 10, 10, 5)]       0
 conv2d_1 (Conv2D)           (None, 8, 8, 4)           184
 conv2d (Conv2D)             (None, 5, 5, 2)           130
 flatten (Flatten)           (None, 50)                0
 dense (Dense)               (None, 1)                 51
=================================================================
Total params: 365 (1.43 KB)
Trainable params: 365 (1.43 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

- The number of CNN weights in the two CNN layers are $(1 + K_1^{(1)} K_2^{(1)} q) q_1 + (1 + K_1^{(2)} K_2^{(2)} q_1) q_2 = 46 \cdot 4 + 65 \cdot 2$.

Pre-trained 2D CNN architectures

- For image classification there are many pre-trained 2D CNN architectures, being trained on a huge data repository from internet to learn to extract local spatial structure.
- We mention AlexNet of Krizhevsky, Sutskever and Hinton (2017) developed in 2012. AlexNet contains 60 million weights that have been pre-trained to extract local structure from images.
- These pre-trained versions can directly be used to extract local structure from images, and a decoder can be fine-tuned on this structure to the specific task at hand; in Zhu and Wüthrich (2021), we provide an example.

1 Convolutional neural networks

2 1D convolutional neural networks

3 2D convolutional neural networks

4 Deep convolutional neural networks

5 Appendix: General purpose layers

Appendix: General purpose layers

Overview

- There are many general purpose layers.
- We have already met the *flatten layer* that brings tensors into vector format.
- The *reshape layer* provides a similar result by allowing to bring data in any possible tensor shape; it is not further discussed here.
- Further special layers of interest are:
  - *Padding layer*
  - *Pooling layer*
  - *Locally-connected network (LCN) layer*

Padding layer

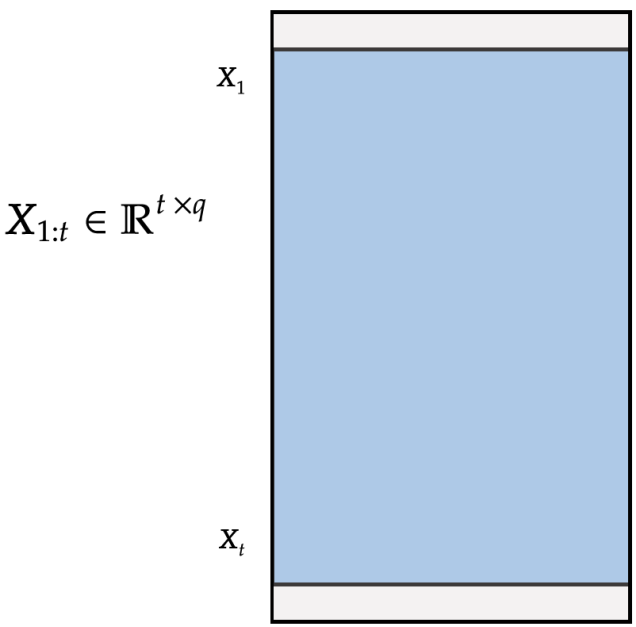
Overview

- A 2D CNN layer typically reduces the spatial dimension from (t, s) to (t', s') because the filter does not fully operate on the edges of the input tensor.
- *Padding* refers to the process of adding extra values, generally zeros, to the edges of a tensor before applying the convolution operation of the CNN layer.
- This technique is used to control the dimension of the output and to ensure that the network effectively captures information located at the edges.

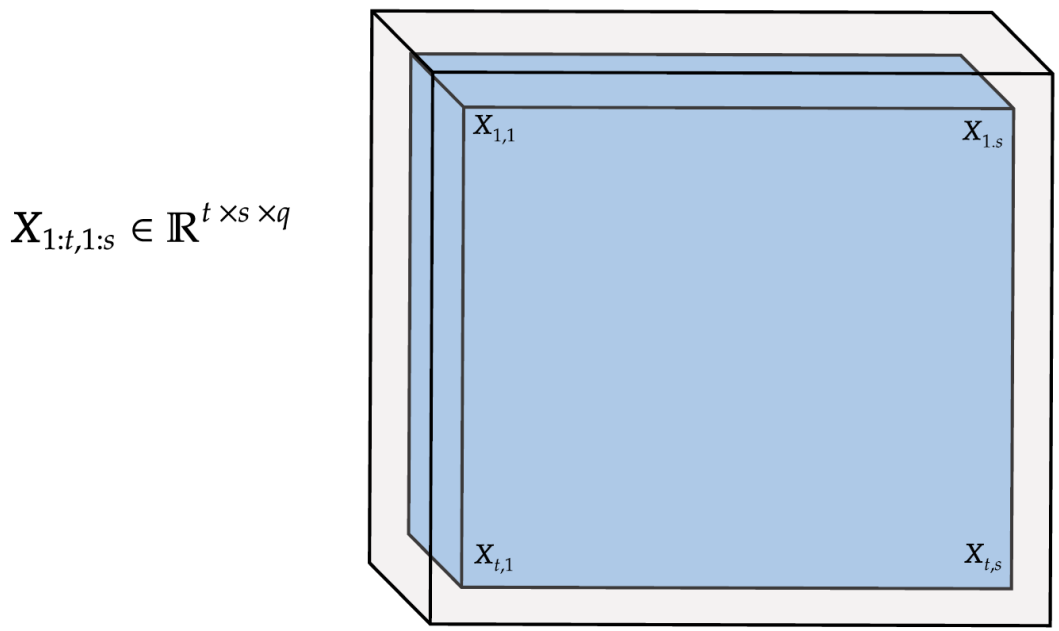
Padding can be applied to various types of input tensors.

- In 1D CNNs (time-series data), padding may be applied at the start and the end of the sequence.
- In 2D CNNs (spatial data), padding is applied along both dimensions, and so on for higher order CNNs.

1D Padding



2D Padding



- The size of padding is typically determined by the filter size K of the CNN layer; we assume stride $\delta = 1$ for the moment.
- For instance, in the case of a 2D CNN layer with a filter of size 3×3, padding of 1 pixel is commonly added to each side of the input to preserve its original dimension.
- Adding one pixel at both ends gives length $1 + t + 1$. This results in

$$t' = \left\lfloor \frac{1 + t + 1 - K_t}{\delta_t} + 1 \right\rfloor = \left\lfloor \frac{1 + t + 1 - 3}{1} + 1 \right\rfloor = t,$$

  and similarly for the columns $s' = \ldots = s$.
- This preserves the spatial shape of the original tensor after applying a 2D CNN layer with the corresponding filter size K and stride δ.

Pooling layer

Overview

- *Pooling layers* are designed to summarize local information and to reduce the sizes of tensors.
- Typically applied after a CNN layer, pooling reduces the tensor's size while preserving its most significant information.
- Pooling layers are similar to CNN layers: both operate on local regions (windows) of the inputs. However, instead of performing a convolution operation, a pooling layer performs a *down-sampling operation* such as the *maximum* or the *averaging operation*.
- Pooling layers require specifying the *pooling size*, which is the analogue to the kernel size in CNN layers, and the *stride*.

- For illustrative purposes, we consider the 1D pooling case, and the 2D pooling case is completely similar.
- For this we select a *pooling size* $K \in \mathbb{N}$ and a *stride* $\delta \in \mathbb{N}$ giving us the mapping

$$\begin{aligned} \boldsymbol{z}^{\text{pool}} \quad : \quad & \mathbb{R}^{t \times q} \to \mathbb{R}^{t' \times q}, \\ & \boldsymbol{X}_{1:t} \mapsto \boldsymbol{z}^{(1)}(\boldsymbol{X}_{1:t}) = \left(z^{\text{pool}}_{u,j}(\boldsymbol{X}_{1:t}) \right)_{1 \le u \le t', 1 \le j \le q}. \end{aligned}$$

- We note that the 1D pooling layer reduces the first dimension of the data from t to $t' = \lfloor \frac{t-K}{\delta} + 1 \rfloor$, while preserving the original covariate dimension (channels) q.
- The standard choice is $K = \delta$ which gives disjoint sets (windows), resulting in the new tensor length

$$t' = \lfloor t/K \rfloor.$$

- The elements of the output depend on the type of pooling used. The most popular pooling layers are the following two.
- *MaxPooling* selects the maximum value from a set of input values in the selected pooling windows ($K = \delta$ here) for $1 \leq j \leq q$

$$z_{u,j}^{\text{pool}}(\boldsymbol{X}_{1:t}) = \max_{(u-1)K+1 \leq k \leq uK} X_{k,j}.$$

- *AveragePooling* computes the average of the values within the given pooling windows ($K = \delta$ here) for $1 \leq j \leq q$

$$z_{u,j}^{\text{pool}}(\boldsymbol{X}_{1:t}) = \frac{1}{K} \sum_{k=(u-1)K+1}^{uK} X_{k,j}.$$

- This extracts either the maximum or the average in each channel $1 \leq j \leq q$ within the disjoint pooling windows labelled by $1 \leq u \leq t'$.

Example of MaxPooling

- The following function, `CNN1Dmax_Model`, defines a 1D CNN architecture that includes a MaxPooling layer. This allows the model to down-sample the feature maps, reducing their size and controlling for over-fitting.

Compared to the previous simpler CNN architecture definition, `CNN1Dmax_Model` requires an additional hyper-parameter:

- `pooling_size`: size of the max-pooling window, the stride is usually set to the same value $\delta = K$.

```
# CNN_1D model with max-pooling layer

CNN1Dmax_Model<- function(seed, input_size, filters, kernel_size, stride,
↪  pooling_size,  output_size) {
      k_clear_session(); set.seed(seed); set_random_seed(seed)
      Design <- layer_input(shape = input_size, dtype = 'float32')

      # 1D CNN encoder
      Response <- Design %>%
         layer_conv_1d(filters = filters, stride = stride, kernel_size =
         ↪  kernel_size, activation = 'tanh') %>%
         # max-pooling layer
         layer_max_pooling_1d(pooling_size) %>%
         # FNN decoder
         layer_flatten() %>%
         layer_dense(output_size, activation = "sigmoid")

      # model definition
      keras_model(inputs = c(Design), outputs = Response)
}
```

- The following code creates an architecture containing a single filtered 1D CNN layer followed by a MaxPooling layer with pooling size equal to 2:

```
model <- CNN1Dmax_Model(
  seed = 100,
  input_size = c(10, 5),
  filters = 1,
  kernel_size = 3,
  stride = 1,
  pooling_size = 2,
  output_size = 1)
```

- This MaxPooling layer shrinks down the size of the decoder FNN layer and we only have 21 weights instead of 25 weights.

```
summary(model)
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 10, 5)]           0
 conv1d (Conv1D)             (None, 8, 1)              16
 max_pooling1d (MaxPooling1D)  (None, 4, 1)            0
 flatten (Flatten)           (None, 4)                 0
 dense (Dense)               (None, 1)                 5
=================================================================
Total params: 21 (84.00 Byte)
Trainable params: 21 (84.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Locally-connected networks

Overview

- A *locally-connected network* (LCN) is a notable variant of CNNs.
- LCNs can be seen as a middle ground between CNNs and FNNs: they preserve the concept of local connectivity while removing the parameter sharing mechanism that characterizes CNNs.
- As a result, in a LCN, each receptive field is assigned a unique set of weights, enhancing the model's flexibility.
- However, this extra flexibility comes at the cost of significantly more parameters than for CNNs.

Definition of a locally-connected network layer

- For simplicity, we use the 1D case for explaining a LCN layer, and the 2D case is completely analogous.
- Formally, a 1D LCN layer with $q_1 \in \mathbb{N}$ filters is a mapping

$$\begin{aligned} \boldsymbol{z}^{(1)} \quad : \quad & \mathbb{R}^{t \times q} \to \mathbb{R}^{t' \times q_1}, \\ & \boldsymbol{X}_{1:t} \mapsto \boldsymbol{z}^{(1)}(\boldsymbol{X}_{1:t}) = \left(z_{u,j}^{(1)}(\boldsymbol{X}_{1:t})\right)_{1 \le u \le t', 1 \le j \le q_1}, \end{aligned}$$

with units

$$z_{u,j}^{(1)}(\boldsymbol{X}_{1:t}) = \phi\left(w_{0,u,j}^{(1)} + \sum_{k=1}^{K} \left\langle \boldsymbol{w}_{k,u,j}^{(1)}, \boldsymbol{X}_{(u-1)\delta+k} \right\rangle\right),$$

with bias terms $w_{0,u,j}^{(1)} \in \mathbb{R}$ and filter weights $\boldsymbol{w}_{k,u,j}^{(1)} \in \mathbb{R}^q$, $1 \le k \le K$, related to the u-th receptive field.

Remarks on LCN layers

- In contrast to a 1D CNN layer, the bias terms and the filter weights have lower indices u corresponding to the u-th receptive field considered. This increases the size of the weights by a factor t' compared to the 1D CNN case because these weights are not shared across $1 \leq u \leq t'$.
- A FNN would have a filter size over the entire input dimension, up to flattening, this turns a LCN layer into a fully-connected FNN layer.

Example of a LCN architecture

- The `LCN1D_Model` function takes the same arguments as the function `CNN1D_Model`.

```
LCN1D_Model <- function(seed, input_size, filters, kernel_size, stride,
↪  output_size) {
      k_clear_session(); set.seed(seed); set_random_seed(seed)
      Design <- layer_input(shape = input_size, dtype = 'float32')

      # 1D LCN
      Response <- Design %>%
        layer_locally_connected_1d(filters = filters, kernel_size = kernel_size,
        ↪  strides = stride, activation = 'tanh') %>%
        layer_flatten() %>%
        layer_dense(units = output_size, activation = "tanh")
      # model definition
      keras_model(inputs = Design, outputs = Response)
}
```

- The following example shows how to create a 1D LCN model with multiple filters $q_1 = 4$, a kernel size of $K = 3$, a stride $\delta = 1$, and 1 output unit.

```
# Example usage
model <- LCN1D_Model(
  seed = 100,
  input_size = c(10, 5),
  filters = 4,
  kernel_size = 3,
  stride = 1,
  output_size = 1)
```

- Compared to the 1D CNN, this LCN has an increased number of weights from 97 to 545, see next table.
- In particular, the 64 shared weights from the CNN layer are now turned into $8 \cdot 64 = 512$ non-shared LCN weights.

```
summary(model)
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 10, 5)]           0
 locally_connected1d (LocallyConne  (None, 8, 4)       512
 cted1D)
 flatten (Flatten)           (None, 32)                0
 dense (Dense)               (None, 1)                 33
=================================================================
Total params: 545 (2.13 KB)
Trainable params: 545 (2.13 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

- This completes the theoretical part on CNNs.

Copyright

- © The Authors
- This notebook and these slides are part of the project "AI Tools for Actuaries". The lecture notes can be downloaded from:

https://papers.ssrn.com/sol3/papers.cfm?abstract_id=5162304

- This material is provided to reusers to distribute, remix, adapt, and build upon the material in any medium or format for noncommercial purposes only, and only so long as attribution and credit is given to the original authors and source, and if you indicate if changes were made. This aligns with the Creative Commons Attribution 4.0 International License CC BY-NC.

References I

Krizhevsky, A., Sutskever, I. and Hinton, G.E. (2017) 'ImageNet classification with deep convolutional neural networks', *Communications of the ACM*, 60(6), pp. 84–90. Available at: https://doi.org/10.1145/3065386.

LeCun, Y. and Bengio, Y. (1998) 'Convolutional networks for images, speech, and time series', in *The Handbook of Brain Theory and Neural Networks*. Cambridge, MA, USA: MIT Press, pp. 255–258. Available at: https://dl.acm.org/doi/10.5555/303568.303704.

Zhu, R. and Wüthrich, M.V. (2021) 'Clustering driving styles via image processing', *Annals of Actuarial Science*, 15(2), pp. 276–290. Available at: https://doi.org/10.1017/S1748499520000317.